A-Z Index • Search

THE SOURCE FOR JAVA™ TECHNOLOGY
java.sun.com

Products & APIs
Developer Connection
Docs & Training
Online Support
Community Discussion
Industry News
Solutions Marketplace
Case Studies

Reflection FREQUENTLY ASKED QUESTIONS

Reflection Home Page | Specification | Reflection API Re ference

What kind of applications might want to use the Reflection API?

The Reflection API is intended for use by tools such as debuggers, class browsers, object inspectors, and interpreters.

When should the Reflection API not be used?

You should avoid the temptation to use the reflection mechanism when other tools more natural to the language would suffice. If you are accustomed to using function pointers in another language, for example, you might think that using `Method` objects are a natural replacement, but usually an object-oriented tool, such as an interface that is implemented by objects that perform the needed action, is better. Programs that needlessly use the Reflection API will be more difficult to read, debug, and maintain.

If the constructor of a class is overloaded, does the `Constructor.newInstance()` method use the types of the arguments to select an overloading?

No. The `Constructor.newInstance()` method is always invoked on a specific overloading of the constructor, previously selected by a call to `Class.getConstructor()` or by other means. The Reflection API does not automatically choose between overloadings.

If a class has an overloaded method, does the `Method.invoke()` method use the types of the arguments to select which method will be invoked?

No. `Method.invoke()` is always invoked on a specific overloading of the method, previously selected by a call to `Class.getMethod()` or by other means. The Reflection API does not automatically choose between overloadings.

How does access control apply to the invocation of the `Constructor.newInstance()`, `Method.invoke()`, `Field.get()`,

and `Field.set()` methods?

When one of these methods is called, the Java Virtual Machine (JVM) performs the access checks described in the Java Language Specification (6.6.1), which are usually carried out by the verifier. For example, when `newInstance()` is called, the checks made by the JVM compare the identity of the caller of `newInstance()` with the access permission and identity of the constructor being called. It is as if the caller of `newInstance()` had a statically-compiled call to the selected constructor. The access check takes into account both the accessibility of the constructor itself, and the accessibility of its class.

It seems that `Method.invoke()` sometimes throws an `IllegalAccessException` when invoking a `public` method. What's going on?

It is a common error to attempt to invoke an overridden method by retrieving the overriding method from the target object. This will not always work, because the overriding method will in general be defined in a class inaccessible to the caller. For example, the following code only works some of the time, and will fail when the `target` object's class is too private:

```
void invokeCommandOn(Object target, String command) {
  try {
    Method m = target.getClass().getMethod(command, new Cla
    m.invoke(target, new Object[] {});
  } catch ...
}
```

The workaround is to use a much more complicated algorithm, which starts with `target.getClass()` and works up the inheritance chain, looking for a version of the method in an accessible class.

[This page was updated: Monday, 31-Jul-2000 10:58:13 MDT]

Products & APIs | Developer Connection | Docs & Training | Support
Community Discussion | Industry News | Solutions Marketplace | Case Studies

Glossary - Feedback - A-Z Index

For more information on the Java technology
and other software from Sun Microsystems, call:
(800) 786-7638
Outside the U.S. and Canada, dial your country's AT&T Direct Access Number first.